Assignment 1: Shell

Date: March 31th, 2019
Deadline: April 14th, 2019 23:59 CET

You must implement a Unix shell similar to `sh`, `bash` and `zsh`. You can choose which exact features to implement in your shell. Not all features are required to obtain a 10 as final grade. A list of all possible features is listed below.

Your shell may not be based on (or exploit) another shell. For example, `system(3)` is forbidden. You must submit your work as a tarball. You can use `make tarball` to generate a tarball for submission automatically. This assignment is individual; you are not allowed to work in teams. Submissions are made to canvas, where they will be tested automatically.

Getting started

1. Unpack the provided source code archive; then run `make`.
2. Try the generated `42sh` binary and familiarize yourself with its interface. Try entering the following commands at its prompt:

   ```
   echo hello
   sleep 1; echo world
   ls | more
   ```

   The framework already provides a front-end for your shell that reads and parses user input into 'nodes'. To understand the default output, read the file `ast.h`.
3. Familiarize yourself with using `valgrind` to check a program.
4. Read the manual pages for `exit(3)`, `chdir(2)`, `fork(2)`, `execvp(3)`, `waitpid(2)` (and the other `wait` functions), `pipe(2)`, `dup2(2)`, `sigaction(2)`. You will need (most of) these functions to implement your shell.
5. Try and modify the file `shell.c` and see the effects for yourself. This is where you should start implementing your shell. You should not have to modify the front-end (`front.c`) or the parser (`parser/` directory).

Features and grading

Your grade will be 0 if you did not submit your work on time, in an invalid format, or has errors during compilation.

If your submission is valid (on time, in valid format, and compiles), your grade starts from 0, and the following tests determine your grade (in no particular order):

- +1pt if you have submitted an archive in the right format with source code that builds without errors.
- +1pt if your shell runs simple commands properly.
- +1pt if your shell supports the `exit` built-in command.
- +1pt if your shell supports the `cd` built-in command.
- +1.5pt if your shell runs sequences of 3 commands or more properly.
- +1.5pt if your shell runs pipes of 2 simple commands properly.
- -1pt if `valgrind` reports memory errors while running your shell.
- -1pt if `gcc -Wall -Wextra` reports warnings when compiling your code.

- -1pt if your shell fails to print an error message on the standard output when an API function fails.
- -1pt if your shell stops when the user enters Ctrl+C on the terminal, or if regular commands are not interrupted when the user enters Ctrl+C.

The following extra features will be tested to obtain higher grades, but only if you have obtained a minimum of 5 points on the list above already:

- +1pt if your shell runs pipes of more than 2 parts consisting of sequences or pipes of simple commands.
- +1pt if your shell supports redirections.
- +0.5pt if your shell supports detached commands.
- +0.5pt if your shell supports executing commands in a sub-shell.
- +0.5pt if your shell supports the `set` and `unset` built-ins for managing environment variables.
- +0.5pt if you parse the PS1 correct and support at least the host name, user name and current path. For more information see PROMPTING in the `bash(1)` manual page.
- +2pt if your shell supports simple job control: Ctrl+Z to suspend a command group, `bg` to continue a job in the background, and `fg` to recall a job to the foreground.
- -1pt if your source files are not neatly indented or formatted.
- -1pt if any of your source files contains functions that are longer then necessary.

The grade will be maximized at 10, so you do not need to implement all features to get a top grade.

Note

Your shell will be evaluated largely automatically. This means features only get a positive grade if they work perfectly, and there will be no half grade for "effort". Most features listed above are separated into several sub-features which are worth points individually.

Example commands

```
## simple commands:
ls
sleep 5   # must not show the prompt too early
```

```
## simple commands, with built-ins:
mkdir t
cd t
/bin/pwd  # must show the new path
exit 42   # terminate with code
```

```
## sequences:
echo hello; echo world # must print in this order
exit 0; echo fail  # must not print "fail"
```

```
## pipes:
ls | grep t
ls | more    # must not show prompt too early
ls | sleep 5 # must not print anything, then wait
sleep 5 | ls # must show listing then wait
ls /usr/lib | grep net | cut -d. -f1 | sort -u
```

```
## redirects:
>dl1 ls /bin; <dl1 wc -l
>dl2 ls /usr/bin; >>dl1 cat dl2 # append
<dl2 wc -l; <dl1 wc -l # show the sum
>dl3 2>&1 find /var/. # errors redirected
```

```
## detached commands:
sleep 5 &  # print prompt early
{ sleep 1; echo hello }& echo world; sleep 3 # invert output
```

```
## sub-shell:
( exit 0 ) # top shell does *not* terminate
cd /tmp; /bin/pwd; ( cd /bin ); /bin/pwd # "/tmp" twice
```

```
## environment variables
set hello=world; env | grep hello # prints "hello=world"
(set top=down); env | grep top # does not print "top=down"

# custom PATH handling
mkdir /tmp/hai; touch /tmp/hai/waa; chmod +x /tmp/hai/waa
set PATH=/tmp/hai; waa # OK
unset PATH; waa # execvp() reports failure
```

Syntax of built-ins

Built-in: `cd <path>`

Change the current directory to become the directory specify in the argument. Your shell does not need to support the syntax `cd` without arguments like Bash does.

Built-in: `exit <code>`

Terminate the current shell process using the specified numeric code. Your shell does not need to support the syntax `exit` without arguments like Bash does.

Built-in (advanced): `set <var>=<value>`

Set the specified environment variable. Your shell does not need to support the syntax `set` without arguments like Bash does.

Built-in (advanced): `unset <var>` **(optional)**

Unset the specified environment variable.

Error handling

Your shell might encounter two types of error:

- When an API function called by the shell fails, for example `execvp(2)` fails to find an executable program. For these errors, your shell must print a useful error message on its standard error (otherwise you can lose 1pt on your grade). You may/should use the helper function `perror(3)` for this purpose.
- When a command launched by the shell exits with a non-zero status code, or a built-in command encounters an error. For these errors, your shell *may* print a useful indicative message, but this will not be tested.

In any case, your program should not leak resources like leaving file descriptors open or forgetting to wait on child processes.

Some tips about the shell

1. It is not necessary that your shell implements advanced features using '*', '?', or '~'.
2. If you do not know how to start, it is best to first start with simple commands, i.e., the 'command' node type.

```
if (node->type == NODE_COMMAND) {
  char *program = node->command.program;
  char **argv = node->command.argv;
  /* Here comes a good combination of fork and exec */
  ...
}
```

3. A shell usually supports redirections on all places of a simple command; `ls > foo` and `>foo ls` are normally equivalent. However, this shell only supports `>foo ls`.
4. Within a 'pipe' construction, all parts should be forked, even if they only contain built-in commands. This keeps the implementation easier.

```
exit 42 # closes the shell
exit 42 | sleep 1  # exit in sub-shell, main shell remains

cd /tmp # changes the directory
cd /tmp | sleep 1  # change directory in sub-shell
                   # main shell does not
```

About the 'arena_*' functions

You are required to free the memory that you allocate to prevent memory leaks. Therefore, at your convenience, we have provided two simple extra libraries: `mc.c` and `arena.c`. You are not required to make use of them, but we personally found them to be very helpful. The main idea behind this setup is that you register with `arena` the memory that you allocate. You also need to supply a function that will free the memory, if that function receives a pointer to it as its sole argument. The freeing function(s) will then automatically be called in the appropriate context, which we will explain later on in a more detailed manner.

At the heart of `arena` lies `mc`, a library that handles memory in a easier way. To start using `mc` you can call `mc_init`, with that return value you can call all `mc_*` functions. These functions enable you to register a tuple of a pointer and function that should be able to free this pointer in the `mc` struct. This way you can deallocate all memory in such a struct by calling `mc_free_all_mem`. Please see the `mc.h` file for more documentation.

The `arena` library uses this function to implement a stack of memory arena's. If we push a new arena, we have a clean space were we can allocate (or register) memory. However as this memory is all saved inside a `mc` we don't have to worry about cleaning it up, we only have to worry about popping our arena's. You might be tempted to think that you never need to pop these arena's, however please note that this is the same as never freeing your memory, which of course is bad practice. So it is good practice to keep these arena's small. Also note that unlike normal memory arena's, this library is not quick, as it uses a linked list of pointers as backing storage. So freeing them is `O(N)` (where `N` is the amount of allocations), so using arena's for small helper functions is probably a bad idea.

If you choose to use the `arena` library, please note there is a `arena_pop_all` function registered for using `atexit(3)`, this means all memory will always be cleaned at the end of your program. As described earlier it is still bad practice to not free old unused memory, however `valgrind` will not show this as an error. If you want to see these errors, to make sure your code is nice and tidy, please change the value of the `dealloc_on_pop_all` variable. If you set this variable to `0`, memory will **NOT** be freed when `arena_pop_all` is called, we will simply remove the pointer to the memory creating a memory leak. A new helper function to make sure memory is freed when `arena_pop_all` is called in child processes but not in the main process is something like this:

```c
pid_t my_fork(void)
{
    pid_t p = fork();
    // In parent proc. it is 0, in child it is 1.
    dealloc_on_pop = !p;
    return p;
}
```

You are allowed to change, add and remove functions from these libraries. However you should make sure that the calls in `front.c` are still valid.

Some tips about the environment

- You can install dependencies on Ubuntu using the following command:

```
sudo apt install build-essential python python-pexpect libreadline-dev \
                 flex valgrind
```

- Use `make local_check` to test your features locally.

- You should also submit your work to our server using `make check` for evaluation. Do so often to make sure your code behaves as expected on our setup, as there may be differences between your local environment and ours. You can submit your work for evaluation as often as you like. This environment is the same as the one used for final grading. **Make sure your submission works on our environment, you will not receive points if it does not.**
- Please report any bugs you may encounter in the automated checking script, such as the awarded points being too high or low. It is strictly prohibited to attempt to cheat the script or attack the infrastructure.
- You are free to choose a C coding style for your code, as long as you are consistent. An example coding style is that of the Linux kernel [1].
- You may add additional source files to organize your code. Add these files to `ADDITIONAL_SOURCES` or `ADDITIONAL_HEADERS` so the environment will correctly use these.

---

1 https://www.kernel.org/doc/html/v4.10/process/coding-style.html